L17: Lessons from Particle System Implementations

Administrative

- Still missing some design reviews
  - Please email to me slides from presentation
  - And updates to reports
  - By Thursday, Apr 16, 5PM
- Grading
  - Lab2 problem 1 graded, problem 2 under construction
  - Return exams by Friday AM
- Upcoming cross-cutting systems seminar,

  Monday, April 20, 12:15-1:30PM, LCR: “Technology Drivers for Multicore Architectures,” Rajeev Balasubramonian, Mary Hall, Ganesh Gopalakrishnan, John Regehr
- Final Reports on projects
  - Poster session April 29 with dry run April 27
  - Also, submit written document and software by May 6
  - Invite your friends! I’ll invite faculty, NVIDIA, graduate students, application owners, ..

Particle Systems

- MPM/GIMP
- Particle animation and other special effects
- Monte-carlo transport simulation
- Fluid dynamics
- Plasma simulations
- What are the performance/implementation challenges?
  - Global synchronization
  - Global memory access costs (how to reduce)
  - Copy to/from host overlapped with computation
- Many of these issues arise in other projects
  - E.g., overlapping host copies with computation image mosaicing

Sources for Today’s Lecture

- A particle system simulation in the CUDA Software Developer Kit called **particles**
- Implementation description in /Developer/CUDA/ projects/particles/doc/particles.pdf
- Possibly related presentation in

  http://www.nvidia.com/content/cudazone/download/ Advanced_CUDA_Training_NVISION08.pdf

  This presentation also talks about finite differencing and molecular dynamics.
- Asynchronous copies in CUDA Software Developer Kit called asyncAPI

Relevant Lessons from Particle Simulation

1. Global synchronization using atomic operation
2. Asynchronous copy from Host to GPU
3. Use of shared memory to cache particle data
4. Use of texture cache to accelerate particle lookup
5. OpenGL rendering

1. Global synchronization

- Concept:
  - We need to perform some computations on particles, and others on grid cells
  - Existing MPM/GIMP provides a mapping from particles to the grid nodes to which they contribute
  - Would like an inverse mapping from grid cells to the particles that contribute to their result
- Strategy:
  - Decompose the threads so that each computes results at a particle
  - Use global synchronization to construct an inverse mapping from grid cells to particles
  - Primitive: ***atomicAdd***

Example Code to Build Inverse Mapping

```
__device__ void addParticleToCell(int3 gridPos, uint
   index, uint* gridCounters, uint* gridCells)
{
    // calculate grid hash
    uint gridHash = calcGridHash(gridPos);

    // increment cell counter using atomics
   int counter = atomicAdd(&gridCounters[gridHash], 1);
   counter = min(counter, params.maxParticlesPerCell-1);
    // write particle index into this cell (very
   uncoalesced!)
    gridCells[gridHash*params.maxParticlesPerCell +
   counter] = index;
}
```

index refers to index of particle

gridPos represents grid cell in 3-d space

gridCells is data structure in global memory for the inverse mapping

What this does:
Builds up gridCells as array limited by max # particles per grid
atomicAdd gives how many particles have already been added to this cell

2. Asynchronous Copy To/From Host

- Warning: I have not tried this, and could not find a lot of information on it.
- Concept:
  - Memory bandwidth can be a limiting factor on GPUs
  - Sometimes computation cost dominated by copy cost
  - But for some computations, data can be "tiled" and computation of tiles can proceed in parallel (some of our projects)
  - Can we be computing on one tile while copying another?
- Strategy:
  - Use page-locked memory on host, and asynchronous copies
  - Primitive **cudaMemcpyAsync**
  - Synchronize with **cudaThreadSynchronize()**

Copying from Host to Device

- cudaMemcpy(dst, src, nBytes, direction)
  - Can only go as fast as the PCI-e bus and not eligible for asynchronous data transfer
- cudaMallocHost(…): Page-locked host memory
  - Use this in place of standard malloc(…) on the host
  - Prevents OS from paging host memory
  - Allows PCI-e DMA to run at full speed
- Asynchronous data transfer
  - Requires page-locked host memory

Example of Asynchronous Data Transfer

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst1, src1, size, dir, stream1);
kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync(dst2, src2, size, dir, stream2);
kernel<<<grid, block, 0, stream2>>>(…);
```

src1 and src2 must have been allocated using cudaMallocHost
stream1 and stream2 identify streams associated with asynchronous call (note 4th “parameter” to kernel invocation)

Particle Data has some Reuse

- Two ideas:
  - Cache particle data in shared memory (3.)
  - Cache particle data in texture cache (4.)

Code from Oster presentation

- **Newtonian mechanics on point masses:**

```
struct particleStruct{
float3 pos;
float3 vel;
float3 force;
};

pos = pos+ vel*dt
vel = vel+ force/mass*dt
```

3. Cache Particle Data in Shared Memory

```
__shared__ float3 s_pos[N_THREADS];
__shared__ float3 s_vel[N_THREADS];
__shared__ float3 s_force[N_THREADS];
int tx= threadIdx.x;
idx= threadIdx.x+ blockIdx.x*blockDim.x;
s_pos[tx] = P[idx].pos;
s_vel[tx] = P[idx].vel;
s_force[tx] = P[idx].force;
__syncthreads();
s_pos[tx] = s_pos[tx] + s_vel[tx] * dt;
s_vel[tx] = s_vel[tx] + s_force[tx]/mass * dt;
P[idx].pos= s_pos[tx];
P[idx].vel= s_vel[tx];
```

4. Use texture cache for read-only data

- Texture memory is special section of device global memory
  - Read only
  - Cached by spatial location (1D, 2D, 3D)
- Can achieve high performance
  - If reuse within thread block so access is cached
  - Useful to eliminate cost of uncoalesced global memory access
- Requires special mechanisms for defining a texture, and accessing a texture

Using Textures: from Finite Difference Example

- Declare a texture ref

```
texture<float, 1, …> fTex;
```

- Bind f to texture ref via an array

```
cudaMallocArray(fArray,…)
cudaMemcpy2DToArray(fArray, f, …);
cudaBindTextureToArray(fTex, fArray…);
```

- Access with array texture functions

```
f[x,y] = tex2D(fTex, x,y);
```

Use of Textures in Particle Simulation

- Macro determines whether texture is used

a. Declaration of texture references in particles_kernel.cu

```
#if USE_TEX
// textures for particle position and velocity
texture<float4, 1, cudaReadModeElementType> oldPosTex;
texture<float4, 1, cudaReadModeElementType> oldVelTex;

texture<uint2, 1, cudaReadModeElementType> particleHashTex;
texture<uint, 1, cudaReadModeElementType> cellStartTex;

texture<uint, 1, cudaReadModeElementType> gridCountersTex;
texture<uint, 1, cudaReadModeElementType> gridCellsTex;
#endif
```

Use of Textures in Particle Simulation

b. Bind/Unbind Textures right before kernel invocation

```
#if USE_TEX
   CUDA_SAFE_CALL(cudaBindTexture(0, oldPosTex, oldPos,
  numBodies*sizeof(float4)));
   CUDA_SAFE_CALL(cudaBindTexture(0, oldVelTex, oldVel,
  numBodies*sizeof(float4)));
#endif

   reorderDataAndFindCellStartD<<< numBlocks, numThreads
  >>>((uint2 *)  particleHash, (float4 *) oldPos, (float4 *) oldVel,
  (float4 *) sortedPos, (float4 *) sortedVel, (uint *)   cellStart);

#if USE_TEX
   CUDA_SAFE_CALL(cudaUnbindTexture(oldPosTex));
   CUDA_SAFE_CALL(cudaUnbindTexture(oldVelTex));
#endif
```

Use of Textures in Particle Simulation

c. Texture fetch (hidden in a macro)

```
ifdef USE_TEX
#define FETCH(t, i) tex1Dfetch(t##Tex, i)
#else
#define FETCH(t, i) t[i]
#endif
```

• Here's an access in particles_kernel.cu

```
float4 pos = FETCH(oldPos, index);
```

5. OpenGL Rendering

- **OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory**
  - **Vertex buffer objects**
  - **Pixel buffer objects**
- **Allows direct visualization of data from computation**
  - **No device to host transfer**
  - **Data stays in device memory –very fast compute / viz**
  - **Automatic DMA from Tesla to Quadro (via host for now)**
- **Data can be accessed from the kernel like any other global data (in device memory)**

OpenGL Interoperability

- Register a buffer object with CUDA
  - **cudaGLRegisterBufferObject(GLuintbuffObj)**;
  - OpenGL can use a registered buffer only as a source
  - Unregister the buffer prior to rendering to it by OpenGL
- Map the buffer object to CUDA memory
  - **cudaGLMapBufferObject(void**devPtr, GLuintbuffObj);**
  - Returns an address in global memory Buffer must be registered prior to mapping
- Launch a CUDA kernel to process the buffer
- Unmap the buffer object prior to use by OpenGL
  - **cudaGLUnmapBufferObject(GLuintbuffObj);**
- Unregister the buffer object
  - **cudaGLUnregisterBufferObject(GLuintbuffObj);**
  - Optional: needed if the buffer is a render target
- Use the buffer object in OpenGL code

Final Project Presentation

- Dry run on April 27
  - Easels, tape and poster board provided
  - Tape a set of Powerpoint slides to a standard 2'x3' poster, or bring your own poster.
- Final Report on Projects due May 6
  - Submit code
  - And written document, roughly 10 pages, based on earlier submission.
  - In addition to original proposal, include
    - Project Plan and How Decomposed (from DR)
    - Description of CUDA implementation
    - Performance Measurement
    - Related Work (from DR)

Final Remaining Lectures

- This one:
  - Particle Systems
- April 20
  - Sorting
- April 22
  - ?
  - Would like to talk about dynamic scheduling?
  - If nothing else, following paper:

"Efficient Computation of Sum-products on GPUs Through Software-Managed Cache," M. Silberstein, A. Schuster, D. Geiger, A. Patney, J. Owens, ICS 2008.

http://www.cs.technion.ac.il/~marks/docs/SumProductPaper.pdf